Programming Languages

2nd edition

Tucker and Noonan

Chapter 8
Semantic Interpretation

To understand a program you must become both the machine and the program.

A. Perlis

Contents

Semantics of a PL

- Defines the meaning of a program
  - *Syntactically valid*
  - *Static type checking valid*

Historical Problem

- Valid program had different meanings on different machines
  - *More than (e.g.) size of an int or float*
- Problem was lack of precision in defining meaning

Methods

- Compiler C on Machine M
  - *Ex: Fortran on IBM 709/7090*
  - *Ex: PL/1 (F) on IBM 360 series*
- Operational Semantics – Ch. 7
- Axiomatic Semantics – Ch. 18
- Denotational Semantics – Ch. 8.4

Example

- Environment
  - *i, j at memory locations 154, 155*

  { <i, 154>, <j, 155> }
- State
  - *i has value 13, j has value -1*

  { ..., <154, 13>, <155, -1>, ...}

Simple State

- Ignore environment
- Set of identifier – value pairs
- Ex: { <i, 13>, <j, -1>
- Special value undefined

8.1 State Transformations

- **Defn**: The *denotational semantics* of a language defines the meanings of abstract language elements as a collection of state-transforming functions.

- **Defn**: A *semantic domain* is a set of values whose properties and operations are independently well-understood and upon which the rules that define the semantics of a language can be based.

Meaningless Program

```
for (i = 1; i > -1; i++)
i--;
// i flips between 0 and 1
// why???
```

Meaningless Expression

- Are all expressions meaningful?
- Give examples

8.2 C++Lite Semantics

- *State* – represent the set of all program states
- A *meaning* function M is a mapping:

  M: Program → State

  M: Statement x State → State

  M: Expression x State → Value

Meaning Rule 8.1

- The meaning of a *Program* is defined to be the meaning of the *body* when given an initial state consisting of the variables of the *decpart* initialized to the *undef* value corresponding to the variable's type.

```
State M (Program p) {
// Program = Declarations decpart; Statement body
return M(p.body, initialState(p.decpart));
}

public class State extends HashMap { ... }
```

```
State initialState (Declarations d) {
State state = new State( );
for (Declaration decl : d)
    state.put(decl.v,  Value.mkValue(decl.t));
}
return state;
}
```

Statements

- M: Statement x State → State

- Abstract Syntax

$$Statement = Skip \mid Block \mid Assignment \mid Loop \mid Conditional$$

```
State M(Statement s, State state) {
if (s instanceof Skip) return M((Skip)s, state);
if (s instanceof Assignment) return M((Assignment)s, state);
if (s instanceof Block) return M((Block)s, state);
if (s instanceof Loop) return M((Loop)s, state);
if (s instanceof Conditional) return M((Conditional)s, state);
throw new IllegalArgumentException( );
}
```

Meaning Rule 8.2

- The meaning of a *Skip* is an identity function
- on the state; that is, the state is unchanged.

- ???

```
State M(Skip s, State state) {
return state;
}
```

Meaning Rule 8.3

- The output state is computed from the input state by replacing the value of the *target* variable by the computed value of the *source* expression.

- Assignment = Variable target;
- Expression source

```
State M(Assignment a, State state) {
return state.onion(a.target, M(a.source, state));
}
// ??? onion
// ??? M(a.source, state)
```

Meaning Rule 8.4

- The meaning of a conditional is:
  - *If the test is true, the meaning of the thenbranch;*
  - *Otherwise, the meaning of the elsebranch*

- Conditional = Expression test;
- Statement thenbranch, elsebranch

```
State M(Conditional c, State state) {
if (M(c.test, state).boolValue( ))
    return M(c.thenbranch);
else
    return M(e.elsebranch, state);
}
```

Expression Semantics

- **Defn**: A *side effect* occurs during the evaluation of an expression if, in addition to returning a value, the expression alters the state of the program.

- Ignore for now.

Expressions

- M: Expression x State → Value

- Expression = Variable | Value | Binary | Unary
- Binary = BinaryOp op; Expression term1, term2
- Unary = UnaryOp op; Expression term
- Variable = String id
- Value = IntValue | BoolValue | CharValue | FloatValue

Meaning Rule 8.7

- The meaning of an expression in a state is a value defined by:

  1. *If a value, then the value. Ex: 3*
  2. *If a variable, then the value of the variable in the state.*
  3. *If a Binary:*

     a) Determine meaning of term1, term2 in the state.

     b) Apply the operator according to rule 8.8

  ...

```
Value M(Expression e, State state) {
if (e instanceof Value)  return (Value)e;
if (e instanceof Variable)  return (Value)(state.get(e));
if (e instanceof Binary) {
    Binary b = (Binary)e;
    return applyBinary(b.op, M(b.term1, state),
        M(b.term2, state);
}
...
```

Dynamically Typed Languages

- Scripting: Perl, Python, PHP
- Object-oriented: Smalltalk, Ruby
- Functional: Scheme, ML, Haskel
- Logic: Prolog
- Our example: dynamically typed C++Lite

```
int main( ) {
   n =3; i =1; f =  1.0;
      while (i < n) {
         i = i + 1;
         f = f * float(i);
      }
}
```

Step	Stmt	n	i	f
1	3	--	--	--
2	4	3	--	--
3	5	3	1	--
4	6	3	1	1.0
5	7	3	1	1.0
6	8	3	2	1.0
7	6	3	2	2.0
8	7	3	2	2.0

Step	Stmt	n	i	f
9	8	3	3	2.0
10	6	3	3	6.0
11	10	3	3	6.0

Perl vs. Python

- Perl: implicit conversions, distinct operators
  - *“2” < “10” : true – numeric comparison*
  - *“2” lt “10” : false – string comparison*
  - *2 lt “10” : false – 2 converted to “2”*
- Python: explicit conversions required
  - *“2” < “10” : false – string comparison*
  - *2 < “10” : error*

Meaning Rule 8.10

- The meaning of a Program is the meaning of its body when given an empty initial state.
  - *Variables declared as encountered*
  - *Type of a variable is type of is value*
  - *In factorial:*
    - i, n – int
    - f – float

C++Dynamic

- Statement = Skip | Block | Assignment | Conditional |

  Loop

  – *Skip, Block unchanged*

  – *Conditional, Loop – check that* test *is bool*

  – *Assignment*

    - add *target* variable to state, if needed
    - no assignment compatibility check needed
    - ???

Meaning Rule 8.11

- The meaning of an expression in the current state is a value defied as follows:
  - *If the expression is a value, then the value itself*
  - *If the expression is a Variable:*
    - If the Variable occurs in the current state, then its associated value.
    - Otherwise the program is meaningless

Meaning Rule 8.11

- *If the expression is a binary:*
  - Determine the value of term1, term2 in current state
  - Apply Rule 4.12 to the operator and values
- *If the expression is a unary:*
  - Determine the value of term in current state
  - Apply Rule 4.13 to the operator and value

- See dynamic-expr.java

Meaning Rule 8.12

- The meaning of a Binary Expression is a Value:
- If operator is arithmetic:
  - *If either operand is an int, both operands must be int; perform int addition for +, int subtraction for -, etc.*
  - *If either operand is a float, both operands must be float; perform float addition for +, float subtraction for -, etc.*
- ...

```
Value M (Expression e, State sigma) {
    if (e instanceof Value)
        return (Value)e;
    if (e instanceof Variable) {
        StaticTypeCheck.check( sigma.containsKey(e),
            "reference to undefined variable");
        return (Value)(sigma.get(e));
    }
```

```
if (e instanceof Binary) {
        Binary b = (Binary)e;
        return applyBinary (b.op,
                M(b.term1, sigma), M(b.term2, sigma));
    }
    if (e instanceof Unary) {
        Unary u = (Unary)e;
        return applyUnary(u.op, M(u.term, sigma));
    }
    throw new IllegalArgumentException(
            "should never reach here");
}
```

```
Value applyBinary (Operator op, Value v1, Value v2)
{
   StaticTypeCheck.check( v1.type( ) == v2.type( ),
                          "mismatched types");
   if (op.ArithmeticOp( )) {
      if (v1.type( ) == Type.INT) {
         if (op.val.equals(Operator.PLUS))
            return new IntValue(
                  v1.intValue( ) + v2.intValue( ));
         if (op.val.equals(Operator.MINUS))
            return new IntValue(
                  v1.intValue( ) - v2.intValue( ));
            ...
```